Integers

CS 220 — Applied Discrete Mathematics

March {24, 26}, 2025

Even and Odd

Definitions (Even, Odd)

An integer is **even** if it is twice some integer.
An integer is **odd** if it is one more than twice some integer.

That is:

$$n \text{ is even} \iff \exists k \in \mathbb{Z},\ n = 2k$$
$$n \text{ is odd} \iff \exists k \in \mathbb{Z},\ n = 2k + 1$$

Lemma (Even-Odd)

If n is odd, then $n + 1$ is even.

Is this “lemma” true? How can you know?

Divisibility

Divisibility

Definition (Divides)

Let $d, n \in \mathbb{Z}$. We say that d **divides** n, written $d \mid n$, if there exists some integer k such that $n = kd$. That is,

$$d \mid n \iff \exists k \in \mathbb{Z},\ n = kd$$

We call d a **factor** of n, and we call n a **multiple** of d.

Facts about Divisibility

- If $a \mid b$ and $a \mid c$, then $a \mid (b + c)$.
- If $a \mid b$ and $k \in \mathbb{Z}$, then $a \mid kb$.
- If $a \mid b$ and $b \mid c$, then $a \mid c$.

Examples

- $3 \mid 6$ and $3 \mid 9$, so $3 \mid 15$
- $5 \mid 10$, so $5 \mid 20$, $5 \mid 30$, etc
- $4 \mid 8$ and $8 \mid 24$, so $4 \mid 24$

Prime and Composite

Definition (Prime, Composite)

Let n be an integer greater than 1. Then n is **prime** if its only positive factors are 1 and n. That is:

$$n \text{ is prime} \iff (n > 1) \wedge (\forall d \in \mathbb{Z}^+,\ d \mid n \Rightarrow (d = 1 \vee d = n))$$

An integer n greater than 1 that is not prime is called **composite**.

Note: 0 and 1 are considered neither prime nor composite.

Examples

- 3 is prime
- 4 is composite (since $2 \mid 4$)
- 5 is prime
- 41 is prime
- 51 is composite (since $3 \mid 51$)
- 61 is prime

Alternative Definitions of Composite

Let $n \in \mathbb{Z}^+$. The following statements are equivalent:

1. n is composite.
2. n has a factor strictly between 1 and n.
   That is, $\exists d \in \mathbb{Z},\ (1 < d < n) \wedge (d \mid n)$.
3. n has a prime factor strictly between 1 and n. ?
   That is, $\exists d \in \mathbb{Z},\ (d \text{ is prime}) \wedge (1 < d < n) \wedge (d \mid n)$.
4. n is the product of two positive integers strictly between 1 and n.
   That is, $\exists a, b \in \mathbb{Z}^+,\ (1 < a < n) \wedge (1 < b < n) \wedge (n = ab)$.

Examples

Consider 12, which is composite.

2. 12 has a factor strictly between 1 and 12 — for example, 6
3. 12 has a prime factor strictly between 1 and 12 — for example, 3
4. 12 is the product of positive integers strictly between 1 and 12 — for example, $3 \cdot 4$, or $2 \cdot 6$

Facts about Prime and Composite Numbers

Theorem

If n is a composite number, then n has a prime divisor $p \leq \sqrt{n}$.

Infinitude of Primes, aka Euclid's Theorem (circa 300 BC)

There is no largest prime number.

That is, there are infinitely many primes.

Prime Factorization

Fundamental Theorem of Arithmetic

Every integer greater than 1 can be written uniquely as the product of primes, where the prime factors are written in order of increasing size.

(A prime may occur more than once in the product.)

For an integer n, this product is called the **prime factorization** of n.
If it includes a prime more than once, we usually write it raised to a power.

Examples

- $15 =$
- $17 =$
- $48 =$
- $100 =$
- $121 =$

Prime Factorization

Fundamental Theorem of Arithmetic

Every integer greater than 1 can be written uniquely as the product of primes, where the prime factors are written in order of increasing size.

(A prime may occur more than once in the product.)

For an integer n, this product is called the **prime factorization** of n.
If it includes a prime more than once, we usually write it raised to a power.

Examples

- $15 = 3 \cdot 5$
- $17 = 17$
- $48 =$
- $100 =$
- $121 =$

Prime Factorization

Fundamental Theorem of Arithmetic

Every integer greater than 1 can be written uniquely as the product of primes, where the prime factors are written in order of increasing size.

(A prime may occur more than once in the product.)

For an integer n, this product is called the **prime factorization** of n.
If it includes a prime more than once, we usually write it raised to a power.

Examples

- $15 = 3 \cdot 5$
- $17 = 17$
- $48 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 3 = 2^4 \cdot 3$
- $100 = 2 \cdot 2 \cdot 5 \cdot 5 = 2^2 \cdot 5^2$
- $121 = 11 \cdot 11 = 11^2$

Greatest Common Divisor (GCD)

Definition (Greatest Common Divisor)

Let $a, b \in \mathbb{Z}+$. The **greatest common divisor** of a and b, written $\gcd(a, b)$, is the greatest $d \in \mathbb{Z}^+$ such that $d \mid a$ and $d \mid b$.

Example

- What is $\gcd(48, 72)$?

  The positive divisors of 48 are $1, 2, 3, 4, 6, 8, 12, 16, 24, 48$.
  The positive divisors of 72 are $1, 2, 3, 4, 6, 8, 9, 12, 18, 24, 36, 72$.
  The common divisors are $1, 2, 3, 4, 6, 8, 12, 24$.
  So $\gcd(48, 72) = 24$.
- What is $\gcd(19, 72)$?

  The positive divisors of 19 are $1, 19$.
  The common divisors are 1.
  So $\gcd(19, 72) = 1$.

Calculating the GCD

Fact about GCD

If $a, b, c \in \mathbb{Z}^+$, then $\gcd(ac, bc) = c \cdot \gcd(a, b)$.

We can use this fact to make a better algorithm for computing the GCD.

Calculating the GCD

Fact about GCD

If $a, b, c \in \mathbb{Z}^+$, then $\gcd(ac, bc) = c \cdot \gcd(a, b)$.

We can use this fact to make a better algorithm for computing the GCD.

Algorithm

To compute $\gcd(a, b)$:

1. Rewrite a and b with their prime factorizations
2. While a and b share a prime factor:
   - Factor it out (with the *minimum* exponent from the two arguments).
   - Repeat with the rest of the prime factorization.
3. When a and b have no prime factors in common, $\gcd(a, b) = 1$. (Why?)

Example

$$\begin{aligned} &\gcd(48, 72) \\ &= \gcd(2^4 \cdot 3^1,\ 2^3 \cdot 3^2) \\ &= 2^3 \cdot \gcd(2^1 \cdot 3^1,\ 3^2) \\ &= 2^3 \cdot 3^1 \cdot \gcd(2^1,\ 3^1) \\ &= 2^3 \cdot 3^1 \cdot 1 = 24 \end{aligned}$$

Later, we'll learn an even better algorithm for GCD.

Relatively Prime

Definition (Relatively Prime)

Two integers a and b are **relatively prime**, aka **coprime**, if $\gcd(a, b) = 1$. That is, they have no positive factor in common other than 1.

Examples

- Are 15 and 28 relatively prime?
- Are 35 and 28 relatively prime?
- Are 55 and 28 relatively prime?

Least Common Multiple

Definition (Least Common Multiple)

Let $a, b \in \mathbb{Z}^+$. The **least common multiple** of a and b, written $\mathrm{lcm}(a, b)$, is the least $n \in \mathbb{Z}^+$ such that $a \mid n$ and $b \mid n$.

Examples

- What is $\mathrm{lcm}(3, 7)$?

  The multiples of 3 are $3, 6, 9, 12, 15, 18, 21, 24, 27, \ldots$.
  The multiples of 7 are $7, 14, 21, 28, 35, 42, 49, \ldots$.
  The common multiples are $21, 42, \ldots$.
  So $\mathrm{lcm}(3, 7) = 21$.
- What is $\mathrm{lcm}(4, 6)$?
  The common multiples are $12, 24, \ldots$. So $\mathrm{lcm}(4, 6) = 12$.
- What is $\mathrm{lcm}(5, 10)$?
  The common multiples are $5, 10, \ldots$. So $\mathrm{lcm}(5, 10) = 10$.

Calculating the LCM

Facts about LCM

- If $a, b, c \in \mathbb{Z}^+$, then $\operatorname{lcm}(ac, bc) = c \cdot \operatorname{lcm}(a, b)$.
- If a and b are relatively prime, then $\operatorname{lcm}(a, b) = a \cdot b$.

Calculating the LCM

Facts about LCM

- If $a, b, c \in \mathbb{Z}^+$, then $\text{lcm}(ac, bc) = c \cdot \text{lcm}(a, b)$.
- If a and b are relatively prime, then $\text{lcm}(a, b) = a \cdot b$.

Algorithm

(Same as GCD algorithm, except #3.)

To compute $\text{lcm}(a, b)$:

1. Rewrite a and b with their prime factorizations
2. Factor out shared prime factors.
3. When a and b have no prime factors in common, $\text{lcm}(a, b) = a \cdot b$.

Example

$$
\begin{aligned}
&\text{lcm}(48, 72) \\
&= \text{lcm}(2^4 \cdot 3^1, 2^3 \cdot 3^2) \\
&= 2^3 \cdot \text{lcm}(2^1 \cdot 3^1, 3^2) \\
&= 2^3 \cdot 3^1 \cdot \text{lcm}(2^1, 3^1) \\
&= 2^3 \cdot 3^1 \cdot (2^1 \cdot 3^1) \\
&= 24 \cdot 6 = 144
\end{aligned}
$$

"Simplified" Algorithms for GCD and LCM

Let $a, b \in \mathbb{Z}^+$. We can compute $\gcd(a, b)$ and $\operatorname{lcm}(a, b)$ as follows:

1. Rewrite a and b as their prime factorizations.
2. Extend the prime factorizations so they use exactly the same set of primes. If a prime was not previously used, its exponent is 0.
3. Then $\gcd(a, b)$ is computed by taking the product of the primes with the **minimum exponent** from both factorizations, and

   $\operatorname{lcm}(a, b)$ is computed by taking the product of the primes with the **maximum exponent** from both factorizations.

Example

$$36 = 2^2 \cdot 3^2 = 2^2 \cdot 3^2 \cdot \mathbf{5^0}$$

$$200 = 2^3 \cdot 5^2 = 2^3 \cdot \mathbf{3^0} \cdot 5^2$$

$$\gcd(36, 200) = 2^{\min(2,3)} \cdot 3^{\min(2,0)} \cdot 5^{\min(0,2)} = 2^2 \cdot 3^0 \cdot 5^0 = 4$$

$$\operatorname{lcm}(36, 200) = 2^{\max(2,3)} \cdot 3^{\max(2,0)} \cdot 5^{\max(0,2)} = 2^3 \cdot 3^2 \cdot 5^2 = 1800$$

Division

Division

Definition (Divisor, Dividend, Quotient, Remainder)

Let $n \in \mathbb{Z}$ and $d \in \mathbb{Z}^+$.
Then there are unique integers q and r with $0 \le r < d$ such that

$$n = qd + r$$

We call n the **dividend**, d the **divisor**, q the **quotient**, and r the **remainder**.

Example

Suppose we divide 17 by 5. We have $17 = \mathbf{3} \cdot 5 + \mathbf{2}$.

That is, 17 is the dividend, 5 is the divisor, 3 is the quotient, and 2 is the remainder.

- "Dividing 17 by 5 produces the quotient 3 with remainder 2."
- "The quotient of 17 divided by 5 is 3."
- "The remainder of 17 divided by 5 is 2."

(It is also true that $17 = 2 \cdot 5 + 7$, but that doesn't satisfy the requirements.)

Examples: Division

Recall: $n = qd + r$ where $0 \leq r < d$.

Examples

- Divide -11 by 3.

  We get $-11 = (-4) \cdot 3 + 1$.
  That is, the quotient is -4 and the remainder is 1.
- Divide 5 by 1.

  We get $5 = 5 \cdot 1 + 0$.
  That is, the quotient is 5 and the remainder is 0.

The Modulo Operator

Definition (Modulo)

Let $a \in \mathbb{Z}$ and let $m \in \mathbb{Z}+$.
Then $a \bmod m$ is the remainder when a is divided by m.

Examples

- $9 \bmod 4 = 1$
- $9 \bmod 3 = 0$
- $9 \bmod 10 = 9$
- $(-13) \bmod 4 = 3$
- $10 \bmod 6 =$
- $12 \bmod 6 =$
- $(-1) \bmod 6 =$
- $(-8) \bmod 6 =$

CS 220 vs Programming Languages

Your favorite programming language may define its “division” and “remainder” or “modulo” operators differently. In CS 220, we use the definition above.
See “Division and Modulus for Computer Scientists” by Daan Leijen for discussion.

The Modulo Operator

Definition (Modulo)

Let $a \in \mathbb{Z}$ and let $m \in \mathbb{Z}+$.
Then $a \bmod m$ is the remainder when a is divided by m.

Examples

- $9 \bmod 4 = 1$
- $9 \bmod 3 = 0$
- $9 \bmod 10 = 9$
- $(-13) \bmod 4 = 3$
- $10 \bmod 6 = 4$
- $12 \bmod 6 = 0$
- $(-1) \bmod 6 = 5$
- $(-8) \bmod 6 = 4$

CS 220 vs Programming Languages

Your favorite programming language may define its "division" and "remainder" or "modulo" operators differently. In CS 220, we use the definition above.
See "Division and Modulus for Computer Scientists" by Daan Leijen for discussion.

Modular Arithmetic

Congruence Modulo m

Definition

Let $a, b \in \mathbb{Z}$ and let $m \in \mathbb{Z}^+$. Then a and b are **congruent modulo m**, written $a \equiv b \pmod{m}$, when m divides their difference. That is:

$$a \equiv b \pmod{m} \iff m \mid (a - b)$$

The integer m is called the **modulus**.

Examples

- $5 \equiv 7 \pmod{2}$ — same parity
- $25 \equiv 95 \pmod{10}$ — same last digit
- $3 \equiv 15 \pmod{12}$ — same clock position
- $-90 \equiv 270 \pmod{360}$ — same angle (in degrees)

Facts about Congruence Modulo m

Let $m \in \mathbb{Z}^+$ and let $a, b, c, d \in \mathbb{Z}$. Then:

Equivalent Definitions

The following statements are equivalent:

- $a \equiv b \pmod{m}$
- $m \mid (a - b)$
- $a \bmod m = b \bmod m$
- there is some $k \in \mathbb{Z}$ such that $a = b + km$

Congruence Properties

If $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$, then

- $a + c \equiv b + d \pmod{m}$
- $a - c \equiv b - d \pmod{m}$
- $a \cdot c \equiv b \cdot d \pmod{m}$

Congruence Modulo m is an Equivalence Relation

Let $m \in \mathbb{Z}^+$. Then $\_ \equiv \_ \pmod{m}$ is an equivalence relation.

Note: By $\_ \equiv \_ \pmod{m}$ I really mean $\{(a, b) \mid a, b \in \mathbb{Z},\ a \equiv b \pmod{m}\}$.

Examples

What are the equivalence classes of $\_ \equiv \_ \pmod{4}$?

Congruence Modulo m is an Equivalence Relation

Let $m \in \mathbb{Z}^+$. Then $\_ \equiv \_ \pmod{m}$ is an equivalence relation.

Note: By $\_ \equiv \_ \pmod{m}$ I really mean $\{(a, b) \mid a, b \in \mathbb{Z},\ a \equiv b \pmod{m}\}$.

Examples

What are the equivalence classes of $\_ \equiv \_ \pmod{4}$?

- $\{\dots, -12, -8, -4, 0, 4, 8, 12, \dots\}$ $\{a \mid a \bmod 4 = 0\}$
- $\{\dots, -11, -7, -3, 1, 5, 9, 13, \dots\}$ $\{a \mid a \bmod 4 = 1\}$
- $\{\dots, -10, -6, -2, 2, 6, 10, 14, \dots\}$ $\{a \mid a \bmod 4 = 2\}$
- $\{\dots, -9, -5, -1, 3, 7, 11, 15, \dots\}$ $\{a \mid a \bmod 4 = 3\}$

Integers and Finite Representations

In computer hardware, we use fixed-size storage to represent integers.
That means we're only representing a proper subset: $Int \subset \mathbb{Z}$.

We need versions of the integer operations that cooperate.

- The true result of the operation on $\mathbb{Z}$ might be too large or too small to represent. This is called **overflow**.
- What should the hardware operations do instead?

What is a good set Int?

What is good behavior for the operations to have?

Integers and Finite Representations

To be concrete, let's limit our representation to *two digits*.
That is, $Int = \{0, 1, \dots, 10, 11, \dots, 98, 99\}$.

Possible modifications to integer operations:

- If a computation overflows, crash (raise exception, or panic, or trap).
- If a computation overflows, produce a special value (Boom).
- Clamp positive overflows to 99, negative overflows to 0.
- Use modular arithmetic with 100 as the modulus.

  That is, overflow results "wrap around".

Properties of Integer Operations

$a + (b + c) = (a + b) + c$	Associativity
$a + (b - c) = (a + b) - c$	Associativity
$a(b + c) = ab + ac$	Distributivity
$a(b - c) = ab - ac$	Distributivity
$0 \cdot a = 0$	Dominance
$a \mid b \;\wedge\; a \mid c \Rightarrow a \mid (b + c)$	DividesSum
$a > 0 \Rightarrow a + b > b$	GreaterSum

Examples: Boom Arithmetic

I'll write $\boxplus$, $\boxminus$, $\boxtimes$ for the operations of "Boom arithmetic".

Examples

$$80 \boxplus (50 \boxminus 40) = 80 \boxplus 10 = 90$$

$$(80 \boxplus 50) \boxminus 40 = \text{Boom} \boxminus 40 = \text{Boom}$$ no Associative

$$10 \boxtimes (30 \boxminus 24) = 10 \boxtimes 6 = 60$$

$$(10 \boxtimes 30) \boxminus (10 \boxtimes 24) = \text{Boom} \boxminus \text{Boom} = \text{Boom}$$ no Distributive

$$0 \boxtimes (50 \boxplus 50) = 0 \boxtimes \text{Boom} = \text{Boom}\ (?)$$ no Dominance

$$3 \mid 99 \;\wedge\; 3 \mid 3, \quad 99 \boxplus 3 = \text{Boom}, \quad 3 \nmid \text{Boom}$$ no DividesSum

$$50 > 0, \quad 50 \boxplus 80 = \text{Boom} \not> 80$$ no GreaterSum

However, if you get a number, you know it's the correct result.

Modular Arithmetic

Definition (Modular Arithmetic)

Let $m \in \mathbb{Z}^+$. Then **modular arithmetic** with m as the **modulus** produces $r \bmod m$ when standard arithmetic produces r. That is, results "wrap around" at m.

I'll write $\boxplus$, $\boxminus$, $\boxtimes$ for modular arithmetic with modulus of 100.
(This notation is not standard.)

Example

$$80 \boxplus (50 \boxminus 40) = 80 \boxplus 10 = 90$$

$$(80 \boxplus 50) \boxminus 40 = 30 \boxminus 40 = 90 \qquad \text{Associative}$$

$$10 \boxtimes (30 \boxminus 24) = 10 \boxtimes 6 = 60$$

$$(10 \boxtimes 30) \boxminus (10 \boxtimes 24) = 0 \boxminus 40 = 60 \qquad \text{Distributive}$$

$$0 \boxtimes (50 \boxplus 50) = 0 \boxtimes 0 = 0 \qquad \text{Dominance}$$

$$3 \mid 99 \ \wedge\ 3 \mid 3, \quad 99 \boxplus 3 = 2, \quad 3 \nmid 2 \qquad \text{no DividesSum}$$

$$50 > 0, \quad 50 \boxplus 80 = 30 \not> 80 \qquad \text{no GreaterSum}$$

Choices: Integer Operation Behavior

Which behavior is better?

Which behavior do programming platforms implement?

Choices: Integer Operation Behavior

Hardware:

- Implements modular arithmetic for integers, but also sets flags (overflow, etc) that can be branched on.
- Implements extended BOOM-like system for floating-point numbers. (Includes $+\infty$, $-\infty$, NaN.)

Choices: Integer Operation Behavior

Hardware:

- Implements modular arithmetic for integers, but also sets flags (overflow, etc) that can be branched on.
- Implements extended BOOM-like system for floating-point numbers. (Includes $+\infty$, $-\infty$, `NaN`.)

LISP/Scheme/Racket:

- Dodges the question by implementing arbitrary-precision integer arithmetic, rational arithmetic, etc.

Choices: Integer Operation Behavior

Hardware:

- Implements modular arithmetic for integers, but also sets flags (overflow, etc) that can be branched on.
- Implements extended Boom-like system for floating-point numbers. (Includes $+\infty, -\infty, \texttt{NaN}$.)

Lisp/Scheme/Racket:

- Dodges the question by implementing arbitrary-precision integer arithmetic, rational arithmetic, etc.

Java:

- Implements modular arithmetic.

Choices: Integer Operation Behavior

Hardware:

- Implements modular arithmetic for integers, but also sets flags (overflow, etc) that can be branched on.
- Implements extended Boom-like system for floating-point numbers. (Includes $+\infty$, $-\infty$, NaN.)

Lisp/Scheme/Racket:

- Dodges the question by implementing arbitrary-precision integer arithmetic, rational arithmetic, etc.

Java:

- Implements modular arithmetic.

C:

- Heh, heh, heh...

Integer Operations in C

What happens on integer overflow?

- for `unsigned` integer types: modular arithmetic
- for `signed` integer types: **undefined behavior**

Undefined behavior:

> *Anything at all can happen; the Standard imposes no requirements. The program may fail to compile, or it may execute incorrectly (either crashing or silently generating incorrect results), or it may fortuitously do exactly what the programmer intended.*

Integer Operations in C

What happens on integer overflow?

- for `unsigned` integer types: modular arithmetic
- for `signed` integer types: **undefined behavior**

Undefined behavior:

> *Anything at all can happen; the Standard imposes no requirements. The program may fail to compile, or it may execute incorrectly (either crashing or silently generating incorrect results), or it may fortuitously do exactly what the programmer intended.*

Up to and including nasal demons (see Jargon File):

> *Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to having demons fly out of your nose.*
>
> *John F. Woods, `comp.std.c`*

Recommended reading:

- "A Guide to Undefined Behavior in C and C++, Part {1,2,3}" by John Regehr

Modular Arithmetic, Refactored

Let's introduce $\mathrm{rep}_{Int} : \mathbb{Z} \to Int$ as the function that takes an integer and returns its equivalent representative in Int:

$$\begin{aligned}\mathrm{rep}_{Int}(a) &= \text{the unique } c \in Int \text{ such that } c \equiv a \pmod{m} \\ &= \text{the unique } c \in Int \text{ such that } c = a + km \text{ for some } k \in \mathbb{Z}\end{aligned}$$

where m is the modulus associated with Int (100 in our running example).

Then $\boxplus, \boxminus, \boxtimes : Int \times Int \to Int$ are simply the following:

$$a \boxplus b = \mathrm{rep}_{Int}(a + b) \qquad a \boxtimes b = \mathrm{rep}_{Int}(a \cdot b)$$

$$a \boxminus b = \mathrm{rep}_{Int}(a - b)$$

That is: calculate the "true result", then find its representative.

Examples: Modular Arithmetic

Let $Int = \{0, 1, 2, \ldots 99\}$, and let $\boxplus$, $\boxminus$, $\boxtimes$ use Int (with $m = 100$).

- $12 \boxplus 23 = \text{rep}_{Int}(35) = 35$
- $60 \boxplus 55 = \text{rep}_{Int}(115) = 15$
- $98 \boxminus 44 = \text{rep}_{Int}(54) = 54$
- $20 \boxminus 37 = \text{rep}_{Int}(-17) = 83$
- $9 \boxtimes 9 = \text{rep}_{Int}(81) = 81$
- $12 \boxtimes 12 = \text{rep}_{Int}(144) = 44$
- $(0 \boxminus 3) \boxtimes 16 = \text{rep}_{Int}(-48) = 52$ *
- $(0 \boxminus 8) \boxtimes 16 = \text{rep}_{Int}(-128) = 72$ *

Examples: Modular Arithmetic

Let $Int = \{0, 1, 2, \ldots 99\}$, and let $\boxplus$, $\boxminus$, $\boxtimes$ use Int (with $m = 100$).

- $12 \boxplus 23 = \text{rep}_{Int}(35) = 35$
- $60 \boxplus 55 = \text{rep}_{Int}(115) = 15$
- $98 \boxminus 44 = \text{rep}_{Int}(54) = 54$
- $20 \boxminus 37 = \text{rep}_{Int}(-17) = 83$
- $9 \boxtimes 9 = \text{rep}_{Int}(81) = 81$
- $12 \boxtimes 12 = \text{rep}_{Int}(144) = 44$
- $(0 \boxminus 3) \boxtimes 16 = \text{rep}_{Int}(-48) = 52$ *
  actually, $= \text{rep}_{Int}(-3) \boxtimes 16 = 97 \boxtimes 16 = \text{rep}_{Int}(1552) = 52$
- $(0 \boxminus 8) \boxtimes 16 = \text{rep}_{Int}(-128) = 72$ *
  actually, $= \text{rep}_{Int}(-8) \boxtimes 16 = 92 \boxtimes 16 = \text{rep}_{Int}(1472) = 72$

Fortunately, we can delay the rep_{Int} to the very end or apply it at each intermediate step — we get the same answer either way! (Why? Review Congruence Properties.)

Representing Negative Integers

We already "have" the negative numbers:

Examples

- $-1 \equiv 99 \pmod{100}$
- $-2 \equiv 98 \pmod{100}$
- $-50 \equiv 50 \pmod{100}$
- ...

- $\text{rep}_{Int}(-1) = 99$
- $\text{rep}_{Int}(-2) = 98$
- $\text{rep}_{Int}(-50) = 50$
- ...

But Int contains no negative numbers, and we never get a "negative result": $0 \boxminus 1 = 99$, not -1.

Representing Negative Numbers

We can keep the basic idea of modular arithmetic with modulus 100 but move the "window" of representative numbers:

Instead of $Int = \{0, 1, 2, \ldots, 99\}$,
let $Int = \{-50, -49, \ldots, -1, 0, 1, 2, \ldots 49\}$.

Definition

The definitions of $\boxplus$, $\boxminus$, $\boxtimes$ are the same, but they refer to the new Int set:

$$a \boxplus b = \text{the unique } c \in Int \text{ such that } c \equiv a + b \pmod{m}$$
$$a \boxminus b = \text{the unique } c \in Int \text{ such that } c \equiv a - b \pmod{m}$$
$$a \boxtimes b = \text{the unique } c \in Int \text{ such that } c \equiv a \cdot b \pmod{m}$$

Or equivalently:

$$\mathrm{rep}_{Int}(a) = \text{the unique } c \in Int \text{ such that } c \equiv a \pmod{m}$$

$$a \boxplus b = \mathrm{rep}_{Int}(a + b) \qquad a \boxminus b = \mathrm{rep}_{Int}(a - b) \qquad a \boxtimes b = \mathrm{rep}_{Int}(a \cdot b)$$

Exercise: Arithmetic with Negative Numbers

Let $Int = \{-50, -49, \ldots, -1, 0, 1, 2, \ldots 49\}$.

- $12 \boxplus 23 =$
- $30 \boxplus 25 =$
- $-20 \boxminus 30 =$
- $-35 \boxminus 45 =$
- $12 \boxtimes 12 =$
- $-3 \boxtimes 16 =$
- $-4 \boxtimes 16 =$

Other Bases

Integers in Other Bases

Definition (Base)

Let b (the **base**) be a positive integer greater than 1.

Then if $n \in \mathbb{Z}^+$, it can be expressed uniquely in the form:

$$n = a_k b^k + a_{k-1} b^{k-1} + \cdots + a_1 b + a_0$$

where k is a nonnegative integer, $a_0, a_1, \ldots, a_k$ are nonnegative integers less than b, and $a_k > 0$.

Then n can be written "in base b" as "$(a_k a_{k-1} \ldots a_1 a_0)_b$".

That is, we use a subscript to indicate the base.
Sometimes we drop the parentheses.

If $n = 0$, then by convention we write "$(0)_b$" or "0_b".

Examples: Bases

Example for $b = 10$ (decimal):

$$\begin{aligned}(859)_{10} &= 8 \cdot 10^2 + 5 \cdot 10^1 + 9 \cdot 10^0 \\ &= 8 \cdot 100 + 5 \cdot 10 + 9 \cdot 1\end{aligned}$$

Example for $b = 2$ (binary):

$$\begin{aligned}(10110)_2 &= 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\ &= 1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 \\ &= 22\end{aligned}$$

Example for $b = 16$ (hexadecimal):

$$\begin{aligned}(3A0F)_{16} &= 3 \cdot 16^3 + 10 \cdot 16^2 + 0 \cdot 16^1 + 15 \cdot 16^0 \\ &= 3 \cdot 4096 + 10 \cdot 256 + 0 \cdot 16 + 15 \cdot 16 \\ &= 14863\end{aligned}$$

In hexadecimal notation, we use letters A to F to indicate numbers 10 to 15.

Example for $b = 64$ (Base64, ignoring padding):

$$\begin{aligned}(\mathrm{Zm9v})_{64} &= 25 \cdot 64^3 + 38 \cdot 64^2 + 61 \cdot 64^1 + 47 \cdot 64^0 \\ &= 25 \cdot 262144 + 38 \cdot 4096 + 61 \cdot 64 + 47 \cdot 1 \\ &= 6713199\end{aligned}$$

$$\begin{aligned}(\mathrm{Zm9v})_{64} &= 25 \cdot 64^3 + 38 \cdot 64^2 + 61 \cdot 64^1 + 47 \cdot 64^0 \\ &= (011001)_2 \cdot (2^6)^3 + (100110)_2 \cdot (2^6)^2 \\ &\quad + (111101)_2 \cdot (2^6)^1 + (101111)_2 \cdot (2^6)^0 \\ &= (011001\ 100110\ 111101\ 101111)_2 \\ &= (01100110\ \ 01101111\ \ 01101111)_2\end{aligned}$$

In Base64, we use A–Z, then a–z, then 0–9, then ...it varies.

Each Base64 character encodes 6 bits, so 4 characters per 3 bytes.

Converting to a Base

Given a base $b \in \mathbb{Z}^+$ and a number $n \in \mathbb{Z}^+$:

- Divide n by b to get the quotient q and remainder r.
- If $q > 0$, recursively convert q to base b to get a "digit" sequence.
  If $q = 0$, then start with the empty sequence.
- Add r to the end of the sequence.

Example

$$\begin{aligned} \text{convert}_4(30) &= \text{convert}_4(7) \mathbin{\|} \text{"2"} & 30 &= 7 \cdot 4 + 2 \\ &= (\text{convert}_4(1) \mathbin{\|} \text{"3"}) \mathbin{\|} \text{"2"} & 7 &= 1 \cdot 4 + 3 \\ &= (\text{"1"} \mathbin{\|} \text{"3"}) \mathbin{\|} \text{"2"} & 1 &= 0 \cdot 4 + 1 \\ &= \text{"132"} \end{aligned}$$

So $30 = (132)_4$.

Arithmetic: Addition

“Long addition” works the same in different bases:

$$
\begin{array}{rr}
 & {\scriptstyle 111} \\
 & 7583_{10} \\
+ & 4932_{10} \\
\hline
 & 12515_{10}
\end{array}
\qquad
\begin{array}{rr}
 & {\scriptstyle 11} \\
 & 1011_{2} \\
+ & 1010_{2} \\
\hline
 & 10101_{2}
\end{array}
\qquad
\begin{array}{rr}
 & {\scriptstyle 111} \\
 & 132_{4} \\
+ & 223_{4} \\
\hline
 & 1021_{4}
\end{array}
$$

For addition of $(x_n \dots x_1 x_0)_b$ and $(y_n \dots y_1 y_0)_b$ with “carries” $(c_{n+1} \dots c_1)$:

- Start at the rightmost (lowest) digit/bit. Define $c_0 = 0$.
- For each k:
  - Compute $s = x_k + y_k + c_k$
  - Divide s by b; set c_{k+1} to the quotient and z_k to the remainder.
- The result is $(c_{n+1} z_n z_{n-1} \dots z_1 z_0)_b$. (Drop leading zero if necessary.)

Digital logic: a “full adder” takes 3 input bits and produces 2 output bits.

Arithmetic: Multiplication

"Long multiplication" works the same in different bases:

$$\begin{array}{rr} & 220_{10} \\ \times & 149_{10} \\ \hline & 1980_{10} \\ & 880_{10} \\ + & 220_{10} \\ \hline & 32780_{10} \end{array} \qquad \begin{array}{rr} & 1011_{2} \\ \times & 101_{2} \\ \hline & 1011_{2} \\ & 0000_{2} \\ + & 1011_{2} \\ \hline & 110111_{2} \end{array}$$

Binary Integers

Processor (CPU, GPU) implementations of integers typically combine

- base-2 representation
- modular arithmetic with modulus of 2^n (for $n \in \{8, 16, 32, 64, \dots\}$)

We'll use 6 bits for our running examples.

- The modulus is $2^6 = 64$.
- $Int = \{-32, \dots, 31\}$
- Let $\boxplus$, $\boxminus$, $\boxtimes$ use Int and modulus 64.

We still need to define:

- a representation strategy mapping Int to sequences of 6 bits that is, a bijection $\mathrm{bin} : Int \to \{0, 1\}^6$
- binary *implementations* of $\boxplus$, $\boxminus$, $\boxtimes$ that satisfy their *specification* in terms of modular arithmetic (ideally, just "long addition", etc)

Representing Integers with Bits

Recall $Int = \{-32, \ldots, 31\}$, and the modulus is 64.
We want $\text{bin} : Int \to \{0, 1\}^6$.

- If n is nonnegative, we'll use its ordinary base-2 representation.
- If n is negative, ...?

Representing Integers with Bits

Recall $Int = \{-32, \dots, 31\}$, and the modulus is 64.
We want $\text{bin} : Int \to \{0,1\}^6$.

- If n is nonnegative, we'll use its ordinary base-2 representation.
- If n is negative, ...?

Let's think about some examples.

- Consider -1. According to the principles of modular arithmetic, it should act the same as 63, since $-1 \equiv 63 \pmod{64}$.
  And we *do* know how to represent 63 using 6 bits: $63 = 111111_2$.
  So let's represent -1 with the bits 111111_2.
- Another example: -12. It should act like 52, since $-12 \equiv 52 \pmod{64}$.
  So we'll represent -12 with the bits 110100_2, since $52 = 110100_2$.
- One more: -32. It should act like 32, since $-32 \equiv 32 \pmod{64}$.
  So we'll represent -32 with the bits 100000_2, since $32 = 100000_2$.

Representing Integers with Bits

Let $UInt = \{0, \ldots, 63\}$. Those are "ordinarily" representable in base 2 using 6 bits. Let $\mathrm{rep}_{UInt} : \mathbb{Z} \to UInt$ be the representative-finder for $UInt$.

$$\begin{aligned}
\mathrm{bin} &: Int \to \{0,1\}^6 \\
\mathrm{bin}(n) &= \mathrm{convert}_2(\mathrm{rep}_{UInt}(n)) \\
&= \begin{cases} \mathrm{convert}_2(n) & \text{if } n \geq 0\text{; that is, } 0 \leq n \leq 31 \\ \mathrm{convert}_2(2^6 + n) & \text{if } n < 0\text{; that is, } -32 \leq n \leq -1 \end{cases}
\end{aligned}$$

This is called the **two's complement** representation of integers.

Examples

$$\begin{aligned}
\mathrm{bin}(0) &= 000000_2 \\
\mathrm{bin}(1) &= 000001_2 \\
\mathrm{bin}(12) &= 001100_2 \\
\mathrm{bin}(31) &= 011111_2
\end{aligned}$$

$$\begin{aligned}
\mathrm{bin}(-1) &= \mathrm{convert}_2(63) = 111111_2 \\
\mathrm{bin}(-12) &= \mathrm{convert}_2(52) = 110100_2 \\
\mathrm{bin}(-31) &= \mathrm{convert}_2(33) = 100001_2 \\
\mathrm{bin}(-32) &= \mathrm{convert}_2(32) = 100000_2
\end{aligned}$$

Arithmetic with Bits

$$
\begin{array}{rr}
 & 17 \\
\boxplus & 12 \\
\hline
 & 29
\end{array}
\qquad
\begin{array}{rr}
 & 010001_2 \\
\boxplus & 001100_2 \\
\hline
 & 011101_2
\end{array}
\qquad\qquad
\begin{array}{rr}
 & 12 \\
\boxplus & -20 \\
\hline
 & -8
\end{array}
\qquad
\begin{array}{rr}
 & \overset{1\,1}{001100_2} \\
\boxplus & 101100_2 \\
\hline
 & 111000_2
\end{array}
$$

$$
\begin{array}{rr}
 & 27 \\
\boxplus & 12 \\
\hline
 & -15
\end{array}
\qquad
\begin{array}{rr}
 & \overset{1\,1}{011011_2} \\
\boxplus & 001100_2 \\
\hline
 & 100111_2
\end{array}
\qquad\qquad
\begin{array}{rr}
 & -30 \\
\boxplus & -20 \\
\hline
 & 14
\end{array}
\qquad
\begin{array}{rr}
 & {\color{red}1}\;100010_2 \\
\boxplus & 101100_2 \\
\hline
 & 001110_2
\end{array}
$$

Addition is just binary “long addition” discarding bits above the 6^{th}.

Topic List

- divides ($a \mid b$), factor, multiple
- prime, composite
- fundamental theorem of arithmetic, prime factorization
- division: dividend, divisor, quotient, remainder
- greatest common divisor, least common multiple ($\gcd, \operatorname{lcm}$)
- relatively prime
- modulo operator ($\bmod$)
- congruence modulo m ($a \equiv b \pmod{m}$)
- representing integers, modular arithmetic
- representing negative integers
- integers in other bases, conversion between bases
- arithmetic (addition, multiplication) in other bases
- two's complement